SOFTWARE PACKAGING AND RELEASE PROCEDURES FOR FREENEST

Marko Silokunnas

Bachelor's thesis
November 2011

Software Engineering
The School of Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) SILOKUNNAS, Marko	Julkaisun laji Opinnäytetyö	Päivämäärä 28.11.2011
	Sivumäärä 61	Julkaisun kieli Englanti
	Luottamuksellisuus ( ) saakka	Verkkojulkaisulupa myönnetty (X)

Työn nimi
SOFTWARE PACKAGING AND RELEASE PROCEDURES FOR FREENEST

Koulutusohjelma
Ohjelmistotekniikka

Työn ohjaaja(t)
KOTKANSALO, Jouko

Toimeksiantaja(t)
HUOTARI, Jouni

Tiivistelmä

Työssä tutkittiin Jyväskylän ammattikorkeakoulun SkyNEST-projektin kehittämää FreeNEST-ohjelmistoa ja sen kehitysprosessia. FreeNEST-ohjelmiston historia ja lähtökohdat kuvailtiin perusteellisesti. Ohjelmiston tämän hetkinen kehitysprosessi kartoitettiin ja paikallistettiin kehitysprosessin ongelmakohdat. Työssä luotiin uusi kehitysprosessi nykyisen kehitysprosessin kartoittamisessa löytyneiden ongelmakohtien pohjalta. Suurin yksittäinen ongelmakohta nykyisessä kehitysmallissa oli virtuaalikoneiden levykuvien käyttö, joka vaikeuttaa hyvien ohjelmistotuotannon kehitystapojen noudattamista.

Pääpaino kehitysprosessin uudelleen luomisessa oli virtuaalikonepohjaisen kehityksen korvaaminen Deb-paketointia käyttävällä järjestelmällä. Uuden kehitysprosessin mahdolliset ongelmakohdat ja haasteet kartoitettiin. Uuden kehitysprosessin ongelmakohdat liittyivät enimmäkseen projektityöntekijöiden kouluttamiseen ja uuden prosessin motiivin ymmärtämiseen. Työssä esitettiin myös toimintamalleja mm. versionhallintaan ja testaukseen. Versionhallinnan ja testauksen toimintamalleilla pyrittiin saamaan mahdollisimman suuri hyöty virtuaalikonepohjaisen kehitysmallin korvaamisesta Deb-paketointia käyttävällä järjestelmällä.

Työ sisältää myös ohjeet Git-versionhallintaohjelmiston käyttöön, Deb-pakettien luomiseen sekä Deb-pakettien jakelupalvelimen pystyttämiseen.

Avainsanat (asiasanat)
FreeNEST, SkyNEST, Debian, Ubuntu, GNU/Linux, Git

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
SILOKUNNAS, Marko	Bachelor´s Thesis	28.11.2011
	Pages 61	Language English
	Confidential ( ) Until	Permission for web publication (X)

Title
SOFTWARE PACKAGING AND RELEASE PROCEDURES PROCEDURES FOR FREENEST

Degree Programme
Software Engineering

Tutor(s)
KOTKANSALO, Jouko

Assigned by
HUOTARI, Jouni

Abstract

This thesis reviews and describes the current development methodology of the FreeNEST project management system developed by the SkyNEST project at JAMK University of Applied Sciences. The thesis briefly explains the goals and history of FreeNEST. The current development methodology is described in detail and its problems and shortcomings are discussed. The thesis proposes a new development methodology based on problems and shortcomings with the current development methodology. The main problem with the current approach is the usage of virtual machine images, which makes it impossible to follow proper software engineering practices.

The main focus of the new development methodology is on replacing the currently used virtual machine images by Deb-packages as the main development medium. Possible problems and challenges for the proposed development methodology are discussed as well. The thesis also proposes guidelines for different tools used in the development of FreeNEST, such as Git. Conventions for version control management and testing are also discussed.

The appendices contain documentation about how to use specific tools mentioned in the thesis, such as Git and git-buildpackage. Documentation about Deb package creation is provided as well.

Keywords
FreeNEST, SkyNEST, GNU/Linux, Debian, Ubuntu, Git

Miscellaneous

CONTENTS

TERMINOLOGY

SkyNEST - A project at JAMK University of Applied Sciences that maintains and develops FreeNEST.

FreeNEST - A project management tool developed by the SkyNEST project at JAMK University of Applied Sciences.

GNU - An operating system that aims to create a free UNIX like operating system. Started by Richard Stallman in 1984. By 1991 all other components of the operating system were finished but GNU was missing a kernel. The word 'free' is meant to mean "Free as in freedom" when speaking about GNU.

Linux - A operating system kernel developed by Linus Torvalds in 1991. Used commonly with GNU to form GNU/Linux distribution packages, such as Debian and Ubuntu.

Virtual machine - *"A virtual machine (VM) is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine."* (Wikipedia, Virtual machine, 16.10.2011)

Dropbox - *"Dropbox is a free service that lets you bring all your photos, docs, and videos anywhere."* (Dropbox Website, 16.10.2011)

Virtual machine image - A computer instance that can be run on a virtual machine. The image can save the state of a virtual computer and can even be transferred to another physical computer running a virtual machine.

Robot Framework - *"Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD)."* (Robot Framework Website, 22.10.2011)

Ixonos Oyj - *"Ixonos Plc (OMX: XNS1V) is a creative mobile solutions company developing wireless technologies, software and solutions for connected devices and mobile services."* (Ixonos, Wikipedia, 27.10.2011)

Scrum - *"Scrum is an iterative, incremental framework for project management often seen in agile software development, a type of software engineering."* (Scrum (development), Wikipedia, 30.10.2011)

TestLink - an open source test management suite. Included in FreeNEST.

Agilo - an open source scrum tool. Included in FreeNEST.

Pulla - *"Pulla is a mildly-sweet Finnish dessert bread"* (Pulla, Wikipedia, 30.10.2011)

Version Control System - Type of programs used widely in software engineering to keep track of changes to a set of files. Using a version control system the developer can follow and revert changes to files. A commonly used abbrevation for ”Version Control System” is VCS.

1 INTRODUCTION

1.1 What is FreeNEST

FreeNEST is developed by the SkyNEST team project at JAMK University of Applied Sciences. FreeNEST is a project management environment that consists of multiple open source software components, which are integrated together to distribute and analyze data across the components. All components developed by the SkyNEST project for FreeNESt are also open source under the BSD license. All modifications to the open source projects are licensed under the same license as the open source project itself.

FreeNEST aims at providing a project enviroment that contains all necessary components for managing the entire lifecycle of a project. Currently FreeNEST is focused on software and hardware projects, but tools for other project types will be added in the future as well.

FreeNEST is meant to be used for a single project and when the project ends, the environment gets taken down. The time it takes to set up an instance of FreeNEST should be as minimal as possible.

History of FreeNEST

FreeNEST has its roots at Ixonos Oyj where a disgruntled testing engineer Marko Rintamäki started a one-man-project called ”Nest”. Nest was a collection of open source software project management tools developed mainly for the needs of Ixonos’s projects. Another engineer at Ixonos, Kai Perälä, helped Rintamäki to compile the first version of Nest. During this time Nest’s design goals according to Peräläs’s master’s thesis (Perälä, K, 2008) were (translated from Finnish by the author of this thesis):

- The system has to allow fast deployment. The tools contained in the system have to be customizable and installable by an expert in a time period of

maximum two days, assuming that the needed equipment is present and working as intended.

- All tools included in the system have to be free of charge and licensed in a manner that allows them to be used commercially.
- The system has to be able to scale up to fill the requirement of different software projects, in other words the usage has to be independent of the project's size, implementation or goals.
- The system has to be able to provide a secure connection over the HTTPS protocol in order to allow remote access.
- The tools included in the system have to support each other. In other words, no program should hinder the usage of other programs.

All versions up to 1.3 were developed while Rintamäki worked at Ixonos. Alongside his work at Ixonos, Rintamäki was also a guest lecturer at JAMK University of Applied Sciences. JAMK got interested in Nest and started a project, named ”SkyNEST”, to faciliate the development of Nest. Rintamäki resigned at Ixonos and moved to JAMK to develop Nest full-time. The SkyNEST project got JAMK accepted into a TEKES funded Cloud Software Program, which *”...aims to significantly improve the competitive position of Finnish software intensive industry in global markets.”*(Cloud Software Program website, 28.10.2011)

The SkyNEST project brought more workforce to the project by allowing students of JAMK to participate in the project. The amount of students working for the project has increased steadily during the course of the project. Version 1.3 was released sometime after a group of students had worked on the project for a few months with Rintamäki. With the 1.3 release Nest was also renamed to FreeNEST.

1.2 Objectives of this thesis

The goal of this thesis is to describe and analyze the current development procedures and methodologies of FreeNEST and provide methods to improve these. The suggested improvements are mainly related to replacing the virtual machine based distribution and development by Debian packages. This thesis also discusses procedures that try to maximise the benefit from the transition to Deb packages. These procedures are mainly related to version control and testing. During the course of writing this thesis a reference implementation of the

proposed development system was created. The creation of this reference implementation is documented in the appendices.

The transition from the current development methodology to the one proposed in this thesis will take a considerable amount of work. It is planned that 1.4 version of FreeNEST will be developed and distributed entirely by the procedures described in this thesis. The planned time for the 1.4 release is somewhere in the middle of 2012.

2 TERMINOLOGY AND THEORETICAL BACKGROUND

2.1 Free Software & Open Source

The term *"Free Software"* does not only refer to the price of the software, but the underlying principles of the software as well. A good definition of Free Software is:

"Free software, software libre or libre software is software that can be used, studied, and modified without restriction, and which can be copied and redistributed in modified or unmodified form either without restriction, or with restrictions that only ensure that further recipients can also do these things and that manufacturers of consumer-facing hardware allow user modifications to their hardware. Free software is generally available without charge, but can have a fee, such as in the form of charging for CDs or other distribution media."(Free Software, Wikipedia, 9.11.2011)

Open Source differs from Free Software mainly by the fact that Open Source emphasises open availability of the source code, not the user's freedom.

All Free Software can considered to be Open Source, but not all Open Source software can be considered to be Free Software.

The ideals behind free software and open source software are enforced through a variety of software licenses, such as the GNU General Public License(GPL), the GNU Lesser General Public License(LGPL) and the BSD License. An American non-profit organization, Free Software Foundation, maintains a list of software licenses that it considers to be Free Software Licences. The Open Source Initiative is an another American organization which maintains a list of software licenses that it considers to be Open Source. Most popular Free & Open Source Software (FOSS) licences are on both lists.

2.2 Debian

According to Debian's website (Debian website, 8.11.2011): *"Debian is a free operating system (OS) for your computer. An operating system is the set of basic programs and utilities that make your computer run. Debian uses the Linux kernel (the core of an operating system), but most of the basic OS tools come from the GNU project; hence the name GNU/Linux"*

Debian is one of the oldest still active GNU/Linux distributions. It is entirely maintained by volunteers. Currently the project has more than one thousand developers. The developers have their own field of expertise to take care of. There are a number of *Package maintainers* whose purpose is to package and upkeep the software packages for the project. There are also people working on documentation, quality assurance, release coordination etc.

Debian as an organization has three foundational documents that define different guidelines for the project. These documents are:

- Debian Social Contract (Debian Social Contract, 9.11.2011)
- Debian Free Software Guidelines (Debian Free Software Guidelines, 9.11.2011)
- Debian constitution (Debian Constitution, 9.11.2011)

Debian Social Contract(DSC) defines broad guidelines and priorities. It mentions that the system should be usable without any non-free components and that the main priorities are the users of the project and free software in general.

Debian Free Software Guidelines(DFSG) is a document that defines what the Debian project considers to be free software. The document also contains examples of licenses that are considered to be compatible with this definition.

Debian Constitution(DC) describes the organizational structure for the project and how decisions are made.

Debian tries to adhere to the free software philosophy as much as possible, but provides optional package repositories for non-free programs as well. The packages contained in the non-free package repositories are considered not to be a part of the Debian project, they are considered to be only packages configured to work with Debian. Support is provided for non-free packages through normal channels (bug tracking system and mailing lists). No non-free components shall be required to run the system.

2.3 Ubuntu

Ubuntu is a GNU/Linux distribution based on Debian. It shares a lot of things with the Debian project, but it is maintained by a company called Canonical Ltd. Canonical was founded by Mark Shuttleworth who used to be a part of the Debian project. Like Debian, Ubuntu puts emphasis on the free software aspects of the project, however it is not as strict about them as Debian.

Ubuntu's focus has always been to create a easy-to-use GNU/Linux distribution. Ubuntu gets released twice a year. Long Term Support(LTS) versions are released every two years.

2.4 Deb packages

Both Ubuntu and Debian share the same packaging format, deb. Deb packages are used to install new software to the operating system. The packages are installed through a tool called apt which downloads the packages from different *package repositories*. The package repositories are collections of packages. Apt also takes care of dependencies between packages.

Users are able to add new package repositories to apt from which to install software that is not packaged by the distribution teams. Such a repository will be created for FreeNEST during the transition from virtual image based development to Debian packages.

Both Ubuntu and Debian have their own package repositories, because even though the format is same the packages are not quaranteed to be binary compatible (all Debian packages cannot be installed to Ubuntu and vice versa).

2.5 Problems with traditional software development

Traditionally, software development has been conducted as a series of phases where each phase might take months or even years to complete. Most traditional software development methodologies contain the following phases:

1. Gather business requirements
2. Design the overall architechture of the product
3. Development

4. Testing and delivery

Most traditional software development methodologies assume that the phases can be conducted strictly in this order and no changes occur. In reality, changes do occur and when they do they usually make a huge mess out of a software project that follows traditional software development methodology. To overcome these problems agile software development methodologies were introduced.

2.6 Agile software development

Agile software development is an umbrella term for many different software development methodologies. In agile software development the project is being worked in small iterations. These iterations are called ”sprints” and usually they take place in a period of 1 to 4 weeks.

While there are numerous different agile software development methodologies most of them put emphasis on the same things. Most agile software development methodologies emphasize teamwork, collaboration and the ability to adapt to different kind of situations quickly. Agile software development methodologies try to fix common problems in traditional software development methodologies.

Few well-known examples of agile software development methodolgies are:

- Scrum
- Extreme Programming
- Feature Driven Development

2.7 Software testing

Software testing aims to provide detailed information whether or not a program fulfills the requirements given to it. Results of software testing can be used to measure quality of the program.

In traditional software development methodologies testing has occured after requirements have been defined and writing the actual code has been completed. Traditionally testing has been conducted entirely by a specially appointed testing team.

Many agile software development methodologies employ *test driven development*, which means that the developers are given some of the responsibilities traditionally given to members of the testing team. Developers are required to write automated test cases for the code they develop and the test cases are automatically run against the code base on a regular basis. After the code has passed the automated tests it will be given to the testing team for additional testing.

Software testing levels according to the SWEBOK (Chapter 5) guide are:

- Unit testing
- Integration testing
- System testing

Definitions for the testing levels according to SWEBOK (Chapter 5):

Unit testing: ”*Unit testing verifies the functioning in isolation of software pieces which are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of tightly related units. A test unit is defined more precisely in the IEEE Standard for Software Unit Testing (IEEE1008-87), which also describes an integrated approach to systematic and documented unit testing. Typically, unit testing occurs with access to the code being tested and with the support of debugging tools, and might involve the programmers who wrote the code.*”(SWEBOK, Chapter 5)

Integration testing: ”*Integration testing is the process of verifying the interaction between software components. Classical integration testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured software.*

Modern systematic integration strategies are rather architecture-driven, which implies integrating the software components or subsystems based on identified functional threads. Integration testing is a continuous activity, at each stage of which software engineers must abstract away lower-level perspectives and concentrate on the perspectives of the level they are integrating. Except for small, simple software, systematic, incremental integration testing strategies are usually preferred to putting all the components together at once, which is pictorially called ”big bang” testing.”(SWEBOK, Chapter 5)

System testing: ” *System testing is concerned with the behavior of a whole system. The majority of functional failures should already have been identified*

during unit and integration testing. System testing is usually considered appropriate for comparing the system to the non-functional system requirements, such as security, speed, accuracy, and reliability. External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level. See the Software Requirements KA for more information on functional and non-functional requirements. "(SWEBOK, Chapter 5)

2.8 Git

According to Git's website : *"Git is a free & open source, distributed version control system designed to handle everything from small to very large projects with speed and efficiency."*(Git website, 8.11.2011)

Git scales extremely well. Git is used for some of the largest FOSS projects on the internet(Linux kernel, X11, Ruby on Rails etc.).

Git contains a feature called *branches* which alongside Git's distributed nature have made Git as popular as it is today. Branches provide a powerful method to manage software development without developers having to worry about messing with someone else's work.

2.8.1 Distributed and centralized version control systems

Distributed version control means that whenever someone clones (in other words *copies*) a Git repository they will receive the entire history of the repository along with their copy. This means that if the central server containing the repository fails for some reason or another everyone who has cloned the repository can restore the entire repository. This also means that once the Git repository has been cloned, it is possible to view any file at any given time from the repository without having to connect to a central server.

Most traditional version control systems (such as SVN and CVS) are centralized. This means that when someone copies the files from the repository they will only receive the current copies of the file. If they want to get an older version of the file they have to contact the centralized server.

2.8.2 Git basics

This section briefly covers a minimum amount of terminology to understand Git's main features. This section will not cover how to actually use Git. The basic usage of Git is covered in Appendix Appendix 1.

As with all version control management systems, Git is used to track changes to a set of files. Git achieves this by storing a snapshot of the files contained in the repository whenever a commit is made. In other words, Git stores entire copies of the files it is tracking (visualized in Figure 2). Naturally, Git does not store copies of all files if only a single file has been changed. If only a single file has been changed, a link to the previous identical file is saved.

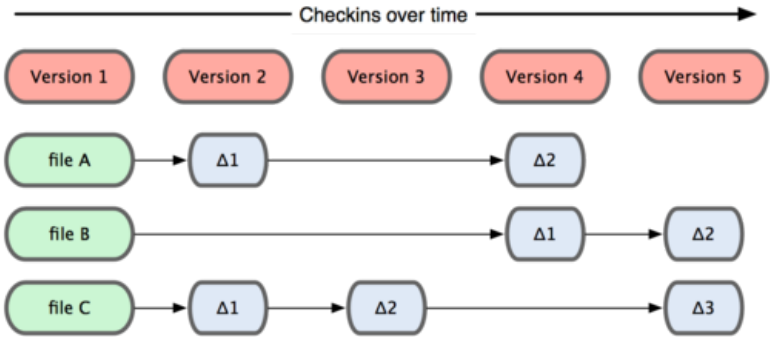

FIGURE 1: How VCSs have handled changes traditionally (Pro Git, chapter 1-3)

This differs from a number of other version control systems which store changes to a set of base files. This is visualized in Figure 1

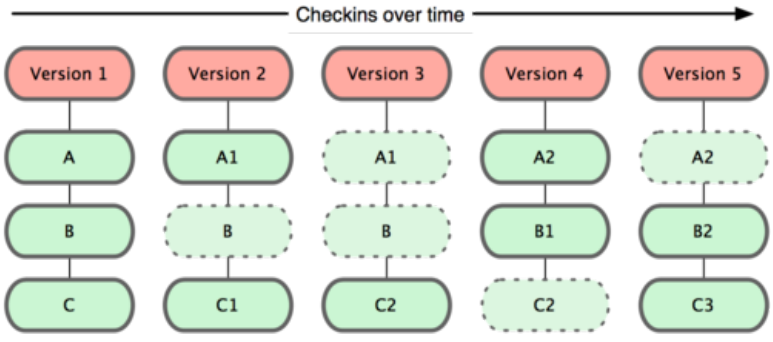

FIGURE 2: How Git handles changes
(Pro Git, 8.11.2011, chapter 1-3)

In Git a commit means a snapshot of the repository. Whenever a commit is made to a Git repository, the current states of all files marked for that commit are stored in the Git database. Git history consists of a series of commits. Commits can be thought of as ”anchor points” in the history of a project and they can be reverted back to, combined etc. Git commits are visualized in Figure 3

FIGURE 3: Git commits

2.8.3 Branches

Branches are Git's way of managing parallel development. A branch is a series of commits in a chronological order. All Git repositories contain at least one branch and it is called *master*.

Branches can be combined by *merging* them. When a branch is merged into another branch, the commits in the source branch will be placed in to the target branch in chronological order. This is visualized in Figure 4. A branch is always associated with a commit that it starts from.

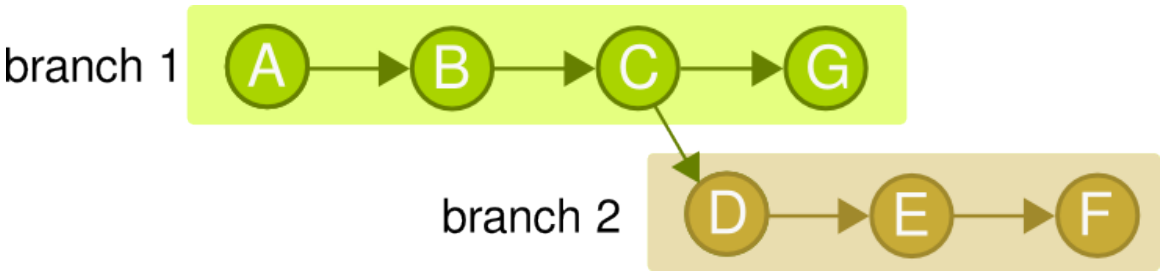

FIGURE 4: Git branches before merging. A is the first commit and G is the last.

Multiple branches can share the same starting commit and changes to the branches will not affect the other branches unless the branches are merged. After a branch has been merged it will still continue to exist and can still be modified. Naturally none of the modifications will affect any other branch unless the branches are merged. A visualization of a simple repository after a merge is provided in Figure 5.

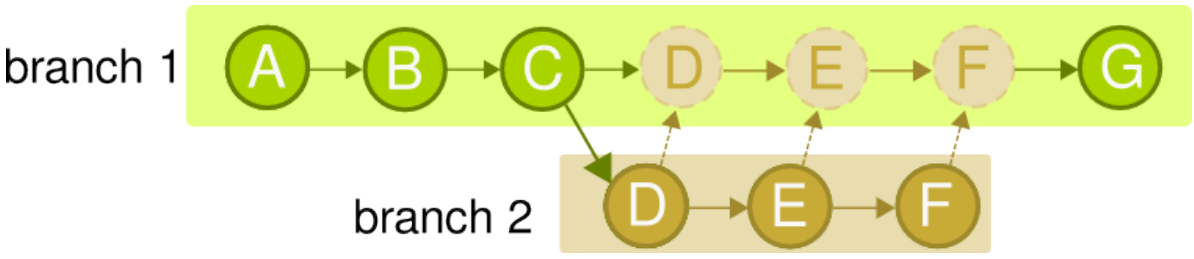

FIGURE 5: Git branches after merge. Note how the order of commits is retained

Branches can be used to efficiently develop individual features without having to

pollute the main branch. This makes sure that the developers can work without any distractions caused by possible conflicts when developers modify the same file. Of course, these conflicts have to be resolved eventually when the branches are merged, but the fact the developers can developed their features without distractions will save time later on, when only a single developer can be assigned to fix the merge conflicts. Branches used for developing a single feature are called *feature branches*.

Branches also enable fine grained control of the history of the main branch. Developers can do their work in their own branches and have that branch as messy as they like. Git allows the developers to tidy up their branches before merging. In this context, tidying up refers to combining small useless commits into larger commits. Clean main branches make progress tracking easier when there are no overhead commits polluting the history.

This kind of working through branches is not possible using traditional version control systems such as Subversion.

3 DEVELOPMENT OF FREENEST

3.1 Project management

Currently Scrum is used as the ”project management framework” for the development of FreeNEST. Length of a sprint is two weeks and daily scrum meetings are used. The daily scrum meetings take place in the morning at 09:30 and everyone should attend. If someone does not attend the daily scrum meeting and does not let someone in the team know in advance, he is required to buy pulla for the rest of the team.

After a sprint a sprint retrospective is held. In the retrospective all team members share what they have done during the sprint and provide suggestions to improve the overall performance of the team.

After the retrospective a sprint planning session is held. The product owner is present in the planning session and provides input about which features should be worked on during the next sprint. The team should be able to discuss the planned features for the next sprint with the product owner and a consensus should be reached. Ideally, during the execution of the sprint the product owner should not be able to change the features which are being worked on, or affect the work of the developers in any way.

After the features have been decided the team estimates the amount of work required by the features. After all features have been estimated the features will be divided into tasks. The tasks are then inserted into Agilo, from where the team members can execute them as they wish. Ideally all tasks should be defined in a manner that all team members are able to execute them. Naturally in real life this is not the case with most tasks, since all of the team members have their own field of expertise and have dedicated a significant amount of time to study it.

3.2 Development methodology of FreeNEST

So far development of FreeNEST has taken place mostly by using only virtual machine images. A few slightly different approaches have been used. These approaches are described below.

Development of FreeNEST started as a virtual machine that had a Linux distribution (Ubuntu 10.04 LTS) and the FreeNEST software components installed. Whenever a new version of a program needed to be installed or some program needed modifications, the work would be done directly to the virtual machine image. Whenever a certain number of changes and modifications were finished the image would be labeled as a release. This sort of development methodology was used when FreeNEST was still mostly a one man project. Testing was not systematic and no test cases were utilized during this time.

As the team size grew slightly the development methodology stayed mostly the same. Individual developers would have their own images and one image that was considered to be the current work-in-progress image would be circulated among the developers using an USB stick or Dropbox. Developers would do their work on this image and redistribute it to others.

As more people joined the the project the development methodology did not change drastically. Some minor improvements were introduced, such as introduction of a ”Bleeding Edge” virtual machine. The idea behind Bleeding Edge was that the developers would do their work on their own private images and when the feature they were working on was complete, they would transfer the changes to Bleeding Edge. This transfer could be copying files or just by doing the modifications performed for the private image by hand on the Bleeding Edge image.

What makes this different from the methodology used previously, is that the Bleeding Edge machine was ran on a centralized virtualization server that the

developers could access directly, making distribution of the work-in-progress image pointless. This saved a great amount of time, because the virtual machine image files are quite large (up to 8 gigabytes) and copying them onto an USB stick or to Dropbox takes a considerable amount of time. Also, systematic testing and test cases were introduced in this phase.

Example of current developer workflow:

1. The developer decides a task to work on (fix a bug or add a new feature) by accepting a task on Agilo.
2. S/he starts a new virtual machine on the centralized virtualization server.
3. S/he manually logs in to the virtual machine started previously and finds out its IP address
4. S/he connects to the virtual machine using SSH
5. S/he performs the actual work on the task (do the fixes, test etc)
6. S/he transfers changes to the Bleeding Edge image. Depending on the task this might include manual copying of files or just replicating the steps to produce the wanted outcome that were performed in the previous step.
7. S/he tests for obvious errors on Bleeding Edge.
8. S/he changes status of the task to ”closed” in Agilo

After the task is closed on Agilo the testing team will test that the feature is working as intended or that the bug is fixed. If problems are found the developer is notified immediately and s/he will have to find the time to fix the issue between his other tasks.

3.3 Testing of FreeNEST

When FreeNEST was mostly a one man project the testing was carried out manually without any test plan or test cases. No automation whatsoever was used.

When more people joined the project systematic testing was also introduced. At first only a single tester was in charge of testing. He would write the test cases to TestLink and perform the tests. Automated testing using Robot Framework was also investigated and a few test cases were written, however it was not taken into use. The developer who did the research on Robot Framework and wrote

the initial test cases left the project and due to a lack of resources no one was appointed to continue the work he had started.

Later as more people joined the project a separate testing team was formed. The testing team's responsibility was to write testcases for features currently found in FreeNEST and perform the tests before publishing a new release. Research on Robot Framework was talked about, but due to the lack of resources no one was appointed to actually write the automated tests using Robot Framework. No methodologies nor infrastructure for the automated tests using Robot Framework existed at this time either. There had been discussions about a ”test cloud”, however no definite plans about its design or implementation had been made.

3.4 Problems with the current approach

3.4.1 General problems

In general the biggest problem in the current way FreeNEST is being developed is the virtual machine image based development. It touches all areas of the development of the product and replacing it with a Deb package based model will greatly increase the efficiency of the entire team. It will also make applying of proper software engineering practices to the project significantly easier.

3.4.2 Problems with development

The current development methodology provides numerous problems for the developers and testers. It is also extremely prone to errors.

The fact that development is mostly conducted on virtual machines makes distributing modifications difficult. In the worst case scenario a new instance of FreeNEST has to be started before work can be done. This includes either downloading the image or setting up a new virtual machines on the centralized virtualization server. This phase can take up to 20 minutes of the developer's time. If multiple developers need to create new virtual machine to the centralized virtualization server the process might take even longer.

When a developer specific private instance is set up, performing the actual work can be started. After the task has been finished and the feature can be considered to be ready to be merged to Bleeding Edge this has to be done by manually copying files or by repeating the exact same steps that were done to the devel-

oper specific private instance.

Working like this makes not only the developer's job more difficult than it should be by introducing annoying and time consuming steps, but it also makes change tracking difficult and error prone.

After changes to Bleeding Edge are finished the task that was worked on will be marked as done to Agilo and work on other tasks can continue. Later on a single developer can corrupt the entire image by making a simple mistake and thus destroying the work of the developers who had made their changes before him. If the corruption is not clearly communicated to every one involved, some of the changes might not end up in Bleeding Edge at all, even though they are marked as done in project task management.

The virtual machine image based development makes also version management difficult. Changes and features cannot be tracked by any version control systems, thus it has to be done manually. Trying to make the development process more simultaneous using branched development is also difficult, because merging changes between the branches would have to be done manually.

A fully automated testing enviroment for FreeNEST in its current state is almost impossible due to the fact that setting up an instance requires quite a great amount of human input. Automating this input over a virtual machine is difficult. Additionally the fact that differentiating changes since the previous version programmatically is extremely difficult means that all test cases must be ran for all versions, unless they are hand picked.

The amount of human input needed to setup the enviroment makes automatic building of releases (e.g. nightly builds) impossible. Every time a release is made one of the developers will have to manually merge the changes of other developers, as described earlier.

Even though the Bleeding Edge machine introduced minor improvements to the development process, it has its own problems as well. The main problem lies with the fact that the development on Bleeding Edge has to be carried out by a single developer at a time. During the development of FreeNEST there have been numerous cases in which only a single developer knows how to fix a certain bug and he is absent during the release building procedure. In the worst case scenario this stalls the entire team until the developer in question arrives to do his work.

3.4.3 Problems with distribution

Virtual machine image based distribution provides major problems for FreeNEST. Distributing virtual machine images makes updating an existing project environment to a newer version extremely frustrating. The user has to manually transfer the data files (database dumps, wiki pages and users) to the new instance, or not transfer them at all. From the user's perspective this makes delivering bug fixes a nightmare. Recently a set of scripts was written to ease this transition, however they have not been included in any of publicly released images as of October 2011.

The virtual machine images take a considerable amount of disk space and transferring them takes a considerable amount of bandwidth. This creates quite heavy requirements for the distribution infrastructure. A sudden spike in FreeNEST downloads on the current distribution infrastructure would probably render the entire distribution channel useless. There are no defined set of instruction about what do if the current distribution infrastructure fails due to an hardware failure or due to a lack of pure CPU power.

3.4.4 Problems with testing

Not surprisingly, the virtual machine images pose a problem for testing as well. Mainly the problems are similiar to the ones faced by the developers (downloading and transferring the images, which takes a lot of time), but problems that are specific for the testing exist as well. There are also problems that have nothing to do with the virtual machine images.

In its current form FreeNEST lacks a lot of feature documentation. Feature documentation about features provided by the 3rd party components of the system are desperately needed in order to be able to write test cases that are detailed enough to be able to test the features thoroughly. The feature documentation could also serve as a guide to help the testers perform their tests, without having to constantly guess what they are supposed to do and how to perform the desired action using the application in question.

4 PROPOSED DEVELOPMENT METHODOLOGY

4.1 Methdology

In order to fix the problems presented in the previous chapter major improvements to almost all parts of the project have to be made.

The use of Scrum as the project methodology is a good idea, however its execution has been slacking somewhat. The entire team and product owners have to commit themselves to work according to the rules that have been accepted by everyone involved. All exceptions to these rules should be extremely rare and there should be a definite reason for them.

Developers should add tasks to Agilo for everything they do and they should close and update the tickets diligently. The product owners have to understand that they are not allowed to disturb the developers with new features during the sprint.

4.2 Replacing virtual image based development

Based on the previous chapters, it can be concluded that the virtual image based development has to replaced. It should be replaced with a modular system that uses Deb packages. All current FreeNEST components should be packaged into Deb packages and a base package for FreeNEST specific issues should be made. The FreeNEST base package should contain only the most basic components of FreeNEST and allow an administrator to log in after installation.

Once the base package is installed, more components can be added to the system by installing additional packages. These additional packages allow users to customize their FreeNEST instances to fit their needs by installing only the software they need.

Transition to Deb package based development will improve the overall experience of the users greatly and make development easier. From the user's perspective the entire system will be more easier to maintain, because updates can be installed for individual packages whenever they become available without having to download an entirely new image. From the developer's perspective this makes it easier to apply proper software engineering practices to the project, thus making development less chaotic and annoying.

A list of definitive improvements over the virtual image based development model:

- FreeNEST will be easier to customize
- Bug fixes and new features can be downloaded automatically alongside updates to the operating system itself.
- Proper software engineering practices can be used for development (version control, component specific testing)
- Testing can be automated
- Continuous integration can be used
- Progress is easier to track from the version control system
- The ability to restore earlier versions of modifications can be done more easily

4.3 Feature documentation

Extensive feature documentation should be provided in order to enable everyone in the project to understand what the components do and what they are used for. The feature documentation could also server as a basis for user manuals. Currently FreeNEST contains several features that have been developed during the course of the project, but are documented poorly or not at all.

Writing feature documentation for the components should be a high priority in the project and should be started as soon as possible.

The actual writing should be divided into 3 phases:

1. Describing the features needed fof FreeNEST(writing feature documentation).
2. Writing user manuals for different components of FreeNEST
3. Writing user manuals for FreeNEST specific components and modifications.

Phase 1 is meant to document the features of FreeNEST. The documentation produced in this phase should contain information about what sort of features FreeNEST provides and find out possible overlaps between components. A good example of overlaps between components is that currently FreeNEST contains multiple wikis, even though only one of them should be used.

Writing feature documentation will help the entire team to form a a better image of the entire project and understand it better. The feature documentation produced in this phase also helps evaluating possible components in the future when comparing them to existing ones.

Phase 2 is meant to produce user documentation. This documentation is needed when teaching new users to use FreeNEST. This documentation could potentially be included in FreeNEST itself(in to the wiki for example), to maximise ease of use and reachability for end users. Videos demonstrating the usage of FreeNEST could also be made using the documentation produced in this step as a base.

Phase 3 is meant to produce documentation about modifications and new features added by the FreeNEST developer team. This documentation is extremely important because it contains documentation for the interoperability between components, which is one of the cornerstones of FreeNEST.

All documentation produced in all of the phases will be extremely useful when designing testing. Feature documentation can be used as a starting point to write automated testing for individual components.

4.4 Version control policies

Version control is extremely important in any software development. Efficient version control procedures and policies are something that can can save a great amount of time from developers when they do not have to do the same work multiple times, can go back to a working version etc. Version control also makes following tasks and their completion easier.

The version control system that is currently used for parts of the FreeNEST project is Git and continuing to use it as the version control system is strongly encouraged. Currently only a part of FreeNEST is hosted in a Git repository due to the virtual image based development, which is explained in more detail in the previous chapters.

All of FreeNEST's development will be migrating into more easily manageable chunks in the form of Debian packages and these chunks will be placed into Git. Proper guidelines and policies will be needed to manage the whole project efficiently and enable the developers to do their work without any unnecessary overhead.

4.4.1 Branching

All individual components of the FreeNEST platform will be given their own repository where the development takes place. All repositories are required to contain three branches for every major version of FreeNEST. These branches are *dev*, *testing* and *stable*. The branch names will be prefixed with the FreeNEST version number that the component is meant for. For example for FreeNEST 1.4 all repositories would have to contain the branches: *1.4-dev*, *1.4-testing* and *stable*.

Developers should also use feature branches for the features they are working on. These feature branch names should be prefixed with *feat/*. No development should take place directly in the dev, testing or stable branches. For example, a branch for a feature, called *Bug#1305* should be called *feat/Bug#1305*. The developers are free to work in the feature branches as they please. Once they have finished their work on the feature they should clean up their branch by combining the commits into meaningful entities. The idea is to try to keep the history of the main branches as clean and informatic as possible. After they have cleaned up their branch they can ask the build manager to merge their changes to the dev branch. Developers should also make sure that the package they are working on builds into a Debian package without errors.

When enough features are considered complete and ready for testing, those features are merged into the testing branch from where the automatic tests will be performed. The testing team will also test the version that is in the testing branch. If an error is found, the fix will be developed in the *dev* branch and once finished it will be merged into the testing branch.

When testing is done and the features are considered to be ready for release, they are merged into the stable branch.

4.4.2 Build manager

The build manager has an essential role in the whole process. The build manager decides when the features are ready for testing and when they are ready to be included in the stable branch. The build manager should always be aware of who is doing what and what the current status of the branches is (have tests passed or not, has building of the packages succeeded). To make the job of merging the feature from branch to another as easy as possible it is highly recommended to use a Git tool that supports merge requests, such as Gitorious, to manage the project.

Rules for merging between the main branches for the build manager:

- No work-in-progress changes should be merged into *dev* from the feature branches. When merging into *dev* the package should build succesfully.
- Changes from *dev* can only be merged into *testing* if the package builds succesfully.
- Changes from *testing* into *stable* can only be merged if the package passes the tests proposed for it.

Because the build manager has to be aware of what is happening in the entire project, it is probably a good idea to appoint the Scrum master as the build manager. The Scrum master has to be aware of the wishes of the product owner and steer the project in that direction, this is why combining the roles of build manager and Scrum master is a good idea as long as the team sizes stay relatively small (max. 5 persons + Scrum master).

Visualization of the branching scheme is provided in Figure 6.

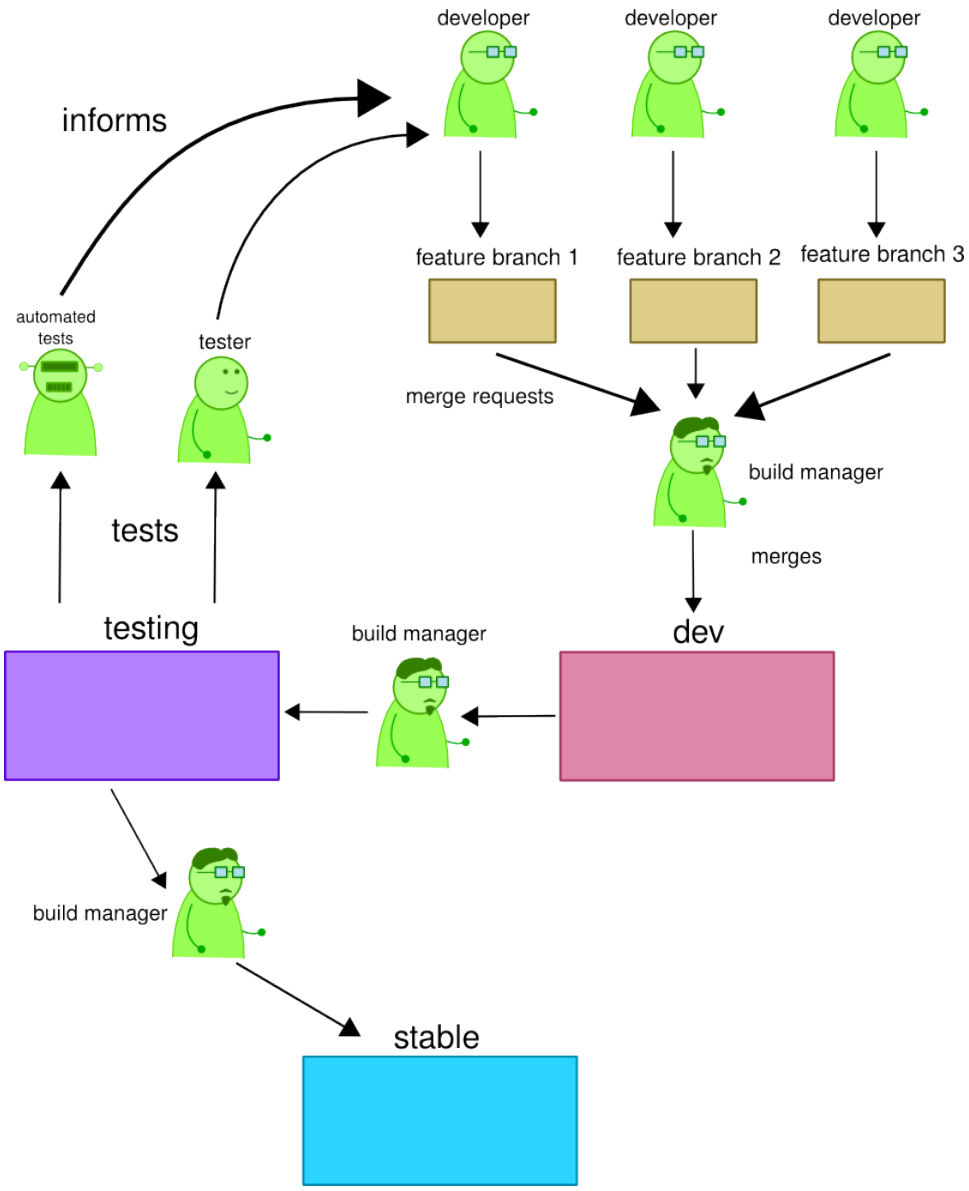

FIGURE 6: Visualization of the branching scheme

4.5 Packaging procedures

4.5.1 General guidelines for packages

All packages should have the proper dependencies set so that they can be installed without having to manually install any other packages. All packages should be packaged in a way that when they are removed they will not leave behind any files that were not in the system before the installation of the package. This rule excludes databases and backups unless the *purge* option is set to true during removal.

FreeNEST packages target mainly Ubuntu and as Ubntu follows Debian's pack-

aging policy, Debian's packaging policy should be used for FreeNEST packages as well.

As the package will be maintained in a Git repository a tool called git-buildpackage should be used to develop the packages. Git-buildpackage allows generation of patches from individual Git commits and makes the job of having the Deb packages in Git easier in general as well.

Building of Deb packages is described in Appendix Appendix 2 and the usage of git-buildpackage is described in Appendix Appendix 3

4.5.2 Package repository

The FreeNEST package repository will have three components which are direct counterparts of the main branches in the Git repository. They are *dev*, *testing* and *stable*.

The state of the branches should directly reflect the repositories. This means that all changes in the *dev* branch of a Git repository should be available in the *dev* package repository shortly after the changes have been merged into the branch. This is achieved by using a build automation tool, such as Jenkins, to compile and upload the packages constantly.

The packages should be built from the branches as soon as possible. If a package's *dev* branch fails to build, the developer responsible for the commit that caused the failure should be notified immediately and the changes should not be merged into the *testing* branch.

Detailed instructions for setting up a Deb package repository are provided in Appendix Appendix 4

4.5.3 Third party libraries in software packages

All packages that need third party libraries should depend on a package that provides that library from the official distribution repositories. If such package does not exist it should be created. When such a package is created, it should be done according to the rules of the distribution and it should be tried to be submitted back to the distribution for inclusion in the official repositories.

4.5.4 Modifications to the software components of FreeNEST

Some modifications will have to be made to almost all of the software components that will be included in FreeNEST. Most software is assumed to be installed manually by the user by downloading a compressed package from the vendor's website and running an installer program. The installer program then usually asks the user for database credentials, timezones etc. The functionality of this installer program will have to be replicated in the package scripts for FreeNEST packages.

In addition to the functiolity of the installer, FreeNESt packages will also have to contain modifications to include the FreeNEST topbar into site templates. The packages will also have to provide some default configuration, such as LDAP authentication settings and database schemas. Database generation of a package should be handled by *dbconfig-common* scripts.

All these modifications should be done to the packages as Quilt patches using git-buildpackage. Detailed usage of git-buildpackage is covered in Appendix Appendix 3.

4.6 Testing procedures

Testing procedures for FreeNEST have improved significantly during the development, however they are still lacking. Automated testing is barely used, as described in chapter 3.4.2 and manual testing is lacking proper testcases in many areas.

Automated testing has to be introduced and writing automated tests for new features should be made mandatory for developers. This can be enforced by the build manager by not accepting merge requests from feature branches unless automated test cases are provided. Specifications for automated tests should be written during sprint planning using feature documentation for that feature.

Before starting to write automated tests, the tests should be designed by referring to feature documentation. Possible similiarities inside the test cases should be found and written down. These similarities should then be used to design an automated testing library for Robot Framework. This task should be assigned to the testing team and the testing team should also be made responsible for keeping automated test cases and the Robot Framework library up to date.

Even though FreeNEST in it's current form contains mostly open source software components written by a third party, automated unit testing can be used.

In the case of FreeNEST units in unit testing can be considered to be individual features of a software component, such as creating a wiki page. These automated tests should be written using Robot Framework. All tests that can be automated should be automated.

The tests that can not be automated should be performed manually by the testing team. No manual testing should take place before merging freatures from the *dev* branch into the *testing* branch. Naturally no features should be merged from *testing* to *stable* before all manual tests pass.

5 RESULTS AND CONCLUSIONS

5.1 Improvements to the current model

The model proposed in this thesis improves the currently used model in numerous ways and even though it is difficult to measure the efficiency of the models scientifically, the proposed model proposes changes to the aspects of the project that have been identified as problematic by the people involved in the project.

As discussed in chapter 3.4 the current model also makes it extremely difficult to follow proper software engineering practices. The new model has been designed to allow the use of good software engineering practices such as automated testing, version control and dividing the system in to small pieces. The proposed model also enhances the usage of agile project management methodologies.

5.2 Possible problems with the proposed model

As stated in the previous chapters, the model proposed in this thesis should greatly improve the efficiency of FreeNEST development. On the other hand, this model has not been been taken into use yet, thus there might be some unknown factors and problems that have not been taken into account in this thesis. If such problems arise, the proposed model should naturally be modified to fit the needs of the project.

Possible problems with the proposed model are:

- Reluctance of current team members to embrace the new methodologies resulting in poorly followed instructions or complete omittance of the in-

structions. This is mainly concerned with the choice of Git as the version control management system and the rules about branching and merging which are rather strict compared to the previous rules.

- Technical problems that slow down development. Examples of possible technical problems include problems with setting up the continuous integration server, creation of Deb packages, following Debian guidelines for web applications.
- Steep learning curve for new developers. New developers of the project need to learn how to use Git and how to create Deb packages. This might take a great amount of time unless the developer is familiar with GNU/Linux and Debian in general.

Team member reluctance to embrace the new methodologies can only be overcome by rationalizing why the proposed system is better than the previous one.

Technical problems are expected but there should be no technical problems that make the usage of the proposed model impossible, because all of the components (Git, Jenkins, Reprepro etc.) used to create this model are widely used and well documented.

5.3 Conclusion

The proposed model will bring much needed improvements to the project, even if not all of the proposed improvements could not be implemented. The single most important improvement is to replace the virtual image based development model with Deb packages. Aside from the fact that developers who are currently working on FreeNEST are familiar with virtual image based development, there is nothing in which the virtual image based development excels at over the Deb package based model.

REFERENCES

Cloud Software Program Website. referenced 28.10.2011.
http://www.cloudsoftwareprogram.org/cloud-program

Debian Constitution. referenced 9.11.2011.
http://www.debian.org/devel/constitution

Debian Free Software Guidelines. referenced 9.11.2011.
http://www.debian.org/social_contract#guidelines

Debian Social Contract. referenced 9.11.2011.
http://www.debian.org/social_contract

Debian website. referenced 8.11.2011. http://www.debian.org

Dropbox website. referenced 22.10.2011. http://www.dropbox.com/tour

Free Software. referenced 9.11.2011. http://en.wikipedia.org/wiki/Free_software

Git website. referenced 8.11.2011. http://git-scm.com/

Ixonos. referenced 27.10.2011. http://en.wikipedia.org/wiki/Ixonos

Linux kernel website. referenced 8.11.2011. http://kernel.org

Perälä, A. 2008. NEST 1.1 - Avoimen Lähdekoodin Projektinhallintaratkaisu.
Pro Gradu -tutkielma. Jyväskylän Yliopisto. Tietotekniikka, ohjelmistotekniikka

Pro Git ebook. referenced 8.11.2011. http://progit.org/book/

Pulla. referenced 30.10.2011. http://en.wikipedia.org/wiki/Pulla

Robot Framework website. referenced 22.10.2011.
http://code.google.com/p/robotframework/

Ruby on Rails website. referenced 8.11.2011.
http://www.rubyonrails.org

Scrum (development). referenced 30.10.2011.
http://en.wikipedia.org/wiki/Scrum_(development)

Software Integration Testing. referenced 14.11.2011.
http://en.wikipedia.org/wiki/Software_testing#Integration_testing

SWEBOK. referenced 14.11.2011.
http://www.computer.org/portal/web/swebok/

Virtual machine. referenced 16.10.2011.
http://en.wikipedia.org/wiki/Virtual_machine.

X11 website. referenced 8.11.2011. http://www.x.org

APPENDICES

Appendix 1 Git basics

Git Basics

This how to describes basic git usage and terms

Terms

Before we go in to detail about how to use Git, we should define what various Git related terms mean.

commits

Git differs from most version control systems by the fact that Git stores entire files instead of changes to files. Whenever you clone a Git repository you also receive a copy of the entire history of the repository. The history contains groups of files saved by Git, these 'groups of files' are called commits in Git

Whenever you make a commit you save the current snapshot of the repository and you can revert back to this snapshot whenever you want. Naturally you can only add specific files to a commit instead of all files in the repository. When you make a commit that changes only a files in the repository only the changed files are compressed and saved in the snapshot. Files that are not changed in a commit will simply be linked to the previous stored version of the file. All changes performed in the files will only be tracked if the changes are included in a commit.

Relevant commands: git commit

staging area

Before you make a commit you have to add the files which you wish to commit to the staging area. The staging area can be considered to be a "loading dock" where you can determine which changes get stored in the repository.

Relevant commands: git add, git reset

branches

Branches are way to manage parallel development in Git. All Git repositories have one branch by default which is called 'master'. You can have many parallel branches that do not affect each other. Branches have a starting point that is a commit. When you create a new branch from a commit the branch will contain the exact same snapshot of the repository than the from which the branch was created. When you make new commits to a branch the changes will be visible only in that branch. You can merge your changes from a branch to another branch. This is extremely useful if you want to just try something experimental and don't want to pollute the main branch. The history of a branch can be altered before the changes are merged into another branch. This makes it easy to keep the main branches clean of useless commits.

Relevant commands: git branch, git checkout

remotes

As with every version control system you have to be able to share your work. In Git this is achieved by a feature called 'remotes'. Remotes are remote Git repositories that you can connect to and copy your commits or branches

to.
Other developers can then download your work from the remote repository.

Relevant commands: git remote, git clone, git push, git fetch, git pull

Basic workflow

A basic Git workflow with an existing repository goes something like this.

1. Perform your changes
2. Add the files you wish to include in the commit to the staging area
3. Inspect the files to make sure what you are going to commit is what you want to commit
4. Commit the changes
5. repeat

A normal workflow that utilizes branches could be as follows:

1. Create a new branch for a feature you are working on
2. Perform your work make a commit etc. Until you consider your feature to be ready
3. Inspect the history of your repository and get rid of useless commits by merging commits in to each other. Try to keep only relevant information in the history.
4. Merge your branch back to the main branch
5. Fix possible merge conflicts

Initializing the repository

Initializing an empty repository

In order to use Git, we have to initialize a Git repository. You can initialize an empty Git repository using

```
$ mkdir repository
$ cd repository
$ git ini
```

This initializes an empty Git repository to a folder called 'repository'.

Cloning an existing repository from an external source

If you want to clone an existing repository from GitHub for example you can use:

```
$ git clone git@github.org:repository.git
```

This will clone the repository called 'repository' from GitHub and create a folder for it in
the current directory

Adding files

In order to make a commit you have to add the files to the staging area as mentioned previously.
You can add files to the staging area using the **git add** command:

```
$ git add file.c
```

You can add all changed files in the repository using

```
$ git add .
```

but this is discouraged because it is then really easy to commit files that you shouldn't have committed.

You can then inspect the staging area using the **git status** command. After the file has been added the output should look like this:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   file.c
```

.gitignore

If you have some files which should never be tracked(such as .swp files generated by Vim, compiled binaries etc.) you can add them to .gitignore. This makes sure Git doesn't tell you that these files have changed and won't add them
to the staging area if you use 'git add .' . You can insert the files you want to ignore to .gitignore one file per line. You
can also use wildcards such as "*.swp" to ignore all files that have a file extension of .swp. The .gitignore file should be placed in to the root directory of the repository and can be committed to the repository if that's what you want.

Example .gitignore file

```
build/*
*.swp
*.tmp
```

Making commits

You can commit the files in your staging area using the git commit command.

```
$ git commi
```

This will open an text editor specified in the EDITOR environment variable. You can describe your commit to in the editor and when you save and quit the commit will be made. If you quit without saving the commit won't be made. This is what the basic commit template file looks like with our example file file.c added:

```

# Please enter the commit message for your changes. Lines starting
```

```
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   file.c
#
```

The first line should be a short description of the commit. Followed by a lengthier explanation of the changes in the following lines.

Example commit file with short and long description:

```
Added file.c to the repository
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   file.c
#
    * This file contains functions that do this and that
```

Save and quit to save the commit.

If you want to use a really simple commit message you can omit opening the EDITOR by specifying the -m flag:

```
$ git commit -m "Really short and simple message"
```

Reverting changes

An integral part of any version control system is the ability to undo your changes. Git naturally provides this functionality.

Fixing mistakes that have not yet been committed

If you have done some modifications that you want to undo, but haven't committed them yet you can do this by executing(this will undo ALL changes since last commit in every file):

```
$ git reset --hard HEAD
```

If you want to undo changes in a single file you can use

```
$ git checkout HEAD file.c
```

This restores the file to the state it was in in the last commit.

Fixing mistakes that have been committed

There are two ways to fix mistakes that have been committed, either by creating a new commit that fixes the mistake or by modifying an existing commit. If you have pushed the change you want to undo to a remote repository you should fix the mistake by making a new commit. This will save you a lot of trouble later on.

Fixing the mistake with a new commit

In order to revert the changes by creating a new commit execute:

```
$ git revert COMMITID
```

COMMITID should naturally be an existing commit. This will create a new commit that undoes the changes performed in the commit. A text editor will be opened for you to input the commit message.

If there are conflicts you are asked to resolve them manually.

Fixing the mistake by modifying an existing commit

If have just forgotten to add a single file to the last commit you can use **git commit --amend** to add a file to the commit. Note that you should not modify any commits that you have pushed to a remote repository, or commits that are merged to other branches!

```
$ git add forgottenfile.c
$ git commit --amend
```

If you need to modify an older commit you have to use **git rebase**. Usage of rebase is not covered in this howto.

Branches

Create new branch

You can create a new branch with

```
$ git branch nameofnewbranc
```

Just creating a new branch does not change your current branch. You have to use

```
$ git checkout nameofnewbranc
```

to change to your new branch. You can also use

```
$ git checkout -b nameofnewbranch
```

which will create the new branch and checkout to it.

All of these methods set the starting point of the new branch to the current commit

Remove branch

You can remove branches with **git branch rm** command

```
$ git branch rm nameofbranch
```

Merging branches

If you want to merge your changes from a branch to another branch you have to use **git merge** command. **git merge** merges changes from the specified branch to your current branch.

```
$ git checkout master # change to master
$ git merge featurebranch1 # merge the changes from featurebranch1 to master
```

If conflicts happen and git can't fix them automagically, you have to resolve them manually
Fix the conflicts manually, add the fixed files and commit once your finished to finish the merge.

Remotes

You can add new remotes with **git remote add** command.

```
$ git remote add github git@github.org:username/repository.git
```

This adds a remote called github that points to a repository on Github.

You can then download branches from the remote using **git fetch** command

```
$ git fetch github
```

The above command downloads all the branches from the remote repository. It does not try to merge anything to the current branch.

If you want to merge the remote changes to the current branch you can use git pull:

```
$ git pull github
```

This will try to merge the changes from an equivalent of your current branch from the remote repository to your local current branch.

Usage of **git fetch** and **git merge** separately is couraged instead of using **git pul**l.

Appendix 2 Debian packaging howto

About this How To

Packaging Debian web applications differs from packaging desktop software quite a bit. Debian project has it's own rules and policies
regarding the packaging of web applications and we should try to follow them as much as possible in order to be able to submit NEST
packages to official Debian repositories some day.

Debian web application packaging policies can be found here
All web applications that need a database should use dbconfig-common to configure, install and update the application databases.

=

=

Basic Web application package without a database

Let's create a package for an extremely simple PHP program called testpackage.

First create a directory for the program and cd into it

```
$ mkdir testpackage
$ cd testpackage
```

Now let's create the program itself, create a file called index.php. it's contents are:

```
<?php
    phpinfo();
?>
```

In order to build a Debian package out of this program, we need to compress it into a .tar.gz package.

```
$ tar -cf testpackage_0.1.orig.tar index.php
$ gzip testpackage_0.1.orig.tar
```

Note that the package name has to be exactly in the form of PACKAGENAME_VERSION.orig.tar.gz for packaging to work.

Next create an empty directory for our package, name of the directory must be in the form of PACKAGENAME-VERSION. Note
that the directory name contains a hyphen instead of a underscore in .orig.tar.gz package name.

```
$ mkdir testpackage-0.1
```

cd into the directory, uncompress the source package and run dh_make

```
$ cd testpackage-0.1
$ tar xf ../testpackage_0.1.orig.tar.gz
$ dh_make
```

You will be prompted for type of the package, choose single binary by typing s.

```
Type of package: single binary, indep binary, multiple binary, library, kernel module,
kernel patch or cdbs?
 [s/i/m/l/k/n/b]
```

You will get an output similar to this:

```
Maintainer name : marko
Email-Address    : marko@unknown
Date               : Fri, 13 May 2011 09:41:27 +0300
Package Name     : testpackage
Version            : 0.1
License            : blank
Using dpatch     : no
Type of Package : Single
Hit <enter> to confirm:
Skipping creating ../testpackage_0.1.orig.tar.gz because it already exists
Currently there is no top level Makefile. This may require additional tuning.
Done. Please edit the files in the debian/ subdirectory now. You should also
check that the testpackage Makefiles install into $DESTDIR and not in / .
```

Now cd into the newly created debian directory

```
$ cd debian
```

We can remove some unnecessary files by issuing the following command.

```
$ rm *.{ex,EX}
```

Now we should edit the control file to match the following

```
Source: testpackage
Section: web
Priority: extra
Maintainer: marko <marko@unknown>
Build-Depends: debhelper (>= 7.0.50~)
Standards-Version: 3.8.4
Homepage: http://www.testpackage.org

Package: testpackage
```

```
Architecture: any
Depends: libapache2-mod-php5 | php5-cgi | php5, ${shlibs:Depends}, ${misc:Depends}
Description: Test package for a simple PHP application
 Simple test package for a simple PHP application that prints out information
 about the server using phpinfo()
```

Note that you should normally fill proper information to the Maintainer line.

Next you should edit the rules file and add the following two lines to the end of the file

```
install:
	dh_install
```

DO NOT JUST COPY&PASTE THESE TWO LINES, write them by yourself and make sure that the second line is intended using a tab not spaces.

Next create a file called install and add the following line

```
*.php		/usr/share/testpackage/www
```

This file tells the package manager to copy all .php files to /usr/share/testpackage/www during installation. Make sure that you separate *.php from /usr/share/testpackage/www using 2 tabs.

Next create a file called postinst with the following contents.

```
#!/bin/sh

set -e

ln -s /usr/share/testpackage/www /var/www/testpackage

#DEBHELPER#

exit 0
```

As you might have noticed this file is a bash script, so you are able to create quite sophisticated installers. This script
gets executed after the package manager has copied the files. In this case we want to create a symbolic link from /var/www/testpackage to /usr/share/testpackage/www so that our newly installed program is available from a browser.

Now let's create another script called postrm. As you might have guessed, this script gets called when we are removing
the package. A logical function for this script is to remove the symbolic we created during installation. To do this we use the following script

```
#!/bin/sh

set -e
```

```
rm /var/www/testpackage

#DEBHELPER#

exit 0
```

Now build the package

```
$ debuild
```

You might get some errors from lintian about malformed maintainer name and template files. You probably will get an error about package signing as well. You can ignore these errors for now, but if you were making a proper package, fixing these would be highly recommended. The package will still work. The next section shows how to fix these errors.

If everything went well, you should now have a .deb file in the folder that contains t*estpackage_0.1.orig.tar.gz*. The package is called *testpackage_0.1-1_i386.deb* and you can install it by typing:

```
$ sudo dpkg -i testpackage_0.1-1_i386.deb
```

Now you should be able to open the application by directing your browser to http://localhost/testpackage

You can remove the package with

```
$ sudo apt-get remove testpackage
```

Creating a package that prompts the user for information during installation

Prompting the user for information during packet installation is extremely useful for a number of reasons. First of all, the installer can pre-configure the program for based on the user's choices in the installer thus saving him/her the trouble of touching any configuration files, creating symlinks and what not.

With Debian packages this sort of functionality is achieved through the use of debconf.

Let's modify the package we made in the previous section so that the user can specify a name for the symlink during installation and it can be changed later by using dpkg-reconfigure.

Let's start by creating a file that contains templates for debconf, the file is called **templates**. Like all files in the previous section, all files in this section are placed in to the **debian** directory.

```
Template: testpackage/symlink_name
Type: string
Default: /var/www/testpackage
Description: Path to this application seen by the user's browser.
 Path to this application as seen by the user's browser. If for example you want this
application to be accesible
 from http://domain.com/IAMAWESOME change this value to IAMAWESOME.
```

The templates file is used to define questions/notices for the user during installation. These templates can even be translated. We won't cover translating template files in this tutorial but you can read more about it here. The link also provides more thorough documentation about the different kinds of template types that debconf can use (string, boolean, select, multiselect...).

First line of the line contains a name for the template which will be used to address this specific template from the installation scripts. Other lines of the file are quite self-explationary. Note that the Description of the template is given in two parts, the part on the first line is a short description for the question and the second is the long description.

Next let's create the script that will actually present the question to the user. This file is called **config**

```
#!/bin/sh -e

# in order to use debconf we have to "source" it. Basically sourcing means
# that the interpreter will execute the specified script.
. /usr/share/debconf/confmodule

# the current value of symlink_name is set to $RET
db_get testpackage/symlink_name

DOCROOT=/var/www

# if the symlink has already been made, remove the old one
if [ -n $RET ]; then
    SYMLINK="$DOCROOT/$RET"
    # make sure that the link exists before removing it
    if [ -e $SYMLINK ]; then
        rm $SYMLINK
    fi
fi
db_input high testpackage/symlink_name || true
db_go
```

After creating the file make it executable by running

```
$ chmod +x ./config
```

Let's rewrite the **postinst** script from the previous section.

```
#!/bin/sh -e



# source debconf
. /usr/share/debconf/confmodule

#get the name set by the user
db_get testpackage/symlink_name
```

```
DOCROOT="/var/www" #document root for the web server
SYMLINK="$DOCROOT/$RET"
PATHTOWWW=/usr/share/testpackage/www

if [ ! -e $SYMLINK ]; then
    ln -s $PATHTOWWW $SYMLINK
fi

#DEBHELPER#

exit 0
```

The behaviour of this hasn't changed from the previous section. It sources debconf and then proceeds to create the symlink. Unlike the previous postinst script, this one make sure that the link doesn't exist before trying to create it. The d**b_get testpackage/symlink_name** retrieves the value set by the user during config phase of the package and sets it to a variable called RET.

Let's proceed to **postrm**.

```
#!/bin/sh -e

# source debconf
. /usr/share/debconf/confmodule

#get the name set by the use
db_get testpackage/symlink_name
DOCROOT="/var/www" #document root for the web server
SYMLINK="$DOCROOT/$RET"
PATHTOWWW=/usr/share/testpackage/www


# check that the symlink exists and remove it if it does
if [ -e $SYMLINK ]; then
    rm $SYMLINK
fi

# remove templates and questions installed by this package
if [ "$1" = "purge" -a -e /usr/share/debconf/confmodule ]; then
    . /usr/share/debconf/confmodule
    db_purge
fi

#DEBHELPER#

exit 0
```

Like in **postinst** , we source **debconf** first. Then we make sure that the link exists before trying to remove it. Finally we make sure that if the package is removed using purge all templates and questions relating to the package are removed as well.

Now you can build the package using **debuild**

```
$ debuild
```

When you install the package for the first time, the installer will ask you to provide a name for the symlink. If you want to change the name, you can do so by executing:

```
$ sudo dpkg-reconfigure testpackage
```

Creating a package with a simple database

Let's create a simple test application that uses a PostgreSQL? database. Write this to a file called **index.php**

```
<?php
    error_reporting( E_ALL );
    // generated during installation of the package
    require_once("../conf/dbconfig.php");

    $connectionString = "";

    if( empty($dbserver) )
    {
        $dbserver = "localhost";
    }
    // for some reason pg_connect requires the host parameter so it has to be
        // included no matter what
    $connectionString .= "host=$dbserver";

    if( !empty($dbname) )
    {
        $connectionString .= " dbname=$dbname";
    }

    if( !empty($dbuser) )
    {
        $connectionString .= " user=$dbuser";
    }

    if( !empty($dbpass) )
    {
        $connectionString .= " password=$dbpass";
    }
```

```
    if( !empty($dbport) )
    {
        $connectionString .= " port=$dbport";
    }

    $dbconn = pg_connect( $connectionString )
             or die("Couldn't connect to database: " . pg_last_error() );


    $query     = 'SELECT * FROM data';
    $result    = pg_query($query ) or die ("Query failed!: " . pg_last_error());

    echo "<table>\n";
    while ( $line = pg_fetch_array( $result, null, PGSQL_ASSOC))
    {
        echo "\t<tr>\n";
        foreach( $line as $col_value )
        {
            echo "\t\t<td>$col_value</td>\n";
        }

        echo "\t</tr>\n";
    }

    echo "</table>\n";

    pg_free_result($result);

    pg_close($dbconn);
?>
```

The connection string is compiled by using values generated during the installation script. The values are written in to the file included by the require_once() line.

Create a folder called **sql** and create a file called **pgsql** with the following contents in to the **sql** directory:

```
CREATE TABLE data
(
    id serial NOT NULL,
    first_name text,
    last_name text,
    CONSTRAINT id PRIMARY KEY (id)
)
WITH (
    OIDS=FALSE
);
```

```
INSERT INTO data (id, first_name, last_name) VALUES (2, 'seppo', 'sepittaja');
INSERT INTO data (id, first_name, last_name) VALUES (1, 'matti', 'meikalainen');
INSERT INTO data (id, first_name, last_name) VALUES (3, 'ville', 'valittaja');
```

This SQL script will be used to generate databases table(s) for the application and fill them with data. The database itself is created by **dbconfig-common**. **dbconfig-common** expects to find this script in a predefined directory with a predefined filename, in order to achieve this we have to copy the script using the **install** file. Contents of the **install** file:

```
*.php        /usr/share/testpackage/www
sql/pgsql    /usr/share/dbconfig-common/data/testpackage/install/
```

Create the necessary folder structure for this application as shown in the previous sections.(create a directory, run dh_make etc...). You should know how to create the **rules** file by now.

control file is as follows:

```
Source: testpackage
Section: web
Priority: extra
Maintainer: marko <marko@unknown>
Build-Depends: debhelper (>= 7.0.50~)
Standards-Version: 3.8.4
Homepage: http://www.testpackage.org

Package: testpackage
Architecture: any
Depends: dbconfig-common, debconf,
          libapache2-mod-php5 | php5-cgi | php5,
          ${shlibs:Depends}, ${misc:Depends}
Description: Testpackage with a database
 An updated version of the extremely useful testpackage!
```

After the necessary files have been copied, using dbconfig-common is extremely easy. You just have source the **dbconfig-common** library after **debconf** and call dbc_go. dbc_go is called in all stages of the installation (config, postinst, prerm and postrm). **dbconfig-common** library will take care of the rest (asking for the database account etc)

Next let's create the config file in to the debian directory.

```
#!/bin/sh

#source debconf
. /usr/share/debconf/confmodule
. /usr/share/dbconfig-common/dpkg/config.pgsql

dbc_go testpackage $@
```

Time for **postinst**

```
#!/bin/sh -e

# source debconf
. /usr/share/debconf/confmodule

DOCROOT="/var/www"
SYMLINK="$DOCROOT/testpackage"
PATHTOWWW="/usr/share/testpackage/www"
CONFDIR="/usr/share/testpackage/conf"
PATHTODBCONF="$CONFDIR/dbconfig.php"

if [ -e $PATHTOWWW ]; then
    ln -s $PATHTOWWW $SYMLINK
fi

#create configuration directory
if [ ! -d $CONFDIR ]; then
    mkdir $CONFDIR
fi

# generate database configuration file for the application
dbc_generate_include="php:$PATHTODBCONF"

# dbconfig-common stuff
. /usr/share/dbconfig-common/dpkg/postinst.pgsql
dbc_go testpackage $@

# set permissions for database configuration so that the web server process
# is able to read them
if [ -e $PATHTODBCONF ]; then
    chown www-data $PATHTODBCONF
fi

#DEBHELPER#

exit 0
```

In order to use information given by the administrator during installation, we have to generate a include file that contains the database information. We can do this by declaring a variable called dbc_generate_include to a value that is of form FORMAT:PATH. Where FORMAT is the format of the configuration file (php, perl etc. For full list of options run 'dbconfig-generate-include --help') and PATH is the absolute path of the configuration file. In the postinst script above we set the configuration to be generated into a PHP script that is located in /usr/share/testpacakge/conf/dbconfig.php . After the variable

has been set we call dbc_go as usual and finally set the permissions for the configuration file so that the webserver process is able to read it.

prerm

```
#!/bin/sh -e

. /usr/share/debconf/confmodule
. /usr/share/dbconfig-common/dpkg/prerm.pgsql

DOCROOT="/var/www" #document root for the web server
SYMLINK="$DOCROOT/testpackage"
PATHTOWWW="/usr/share/testpackage/www"
CONFDIR="/usr/share/testpackage/conf"
PATHTODBCONF="$CONFDIR/dbconfig.php"

# check that the symlink exists and remove it if it does
if [ -e $SYMLINK ]; then
    rm $SYMLINK
fi

# remove database configuration
if [ -e $PATHTODBCONF ]; then
    rm $PATHTODBCONF
fi

# remove configuration directory
if [ -d $CONFDIR ]; then
    rmdir $CONFDIR
fi

dbc_go testpackage $@

#DEBHELPER#

exit 0
```

postrm

```
#!/bin/sh -e

# source debconf
. /usr/share/debconf/confmodule

DOCROOT="/var/www" #document root for the web server
SYMLINK="$DOCROOT/testpackage"
PATHTOWWW="/usr/share/testpackage/www"
```

```
# dbconfig-common stuff
. /usr/share/dbconfig-common/dpkg/postrm.pgsql
dbc_go testpackage $@

# remove templates and questions installed by this package
if [ "$1" = "purge" -a -e /usr/share/debconf/confmodule ]; then
    . /usr/share/debconf/confmodule
    db_purge
fi


#DEBHELPER#

exit 0
```

Appendix 3 git-buildpackage

How to Use Debian Git Buildpackage

This how to will describe how to use Debian's git-buildpackage tool to create a simple testpackage. The testpackage contains a single PHP file that prints out phpinfo(). This how to assumes that you have read HowToDebianWebApplicationPackaging.

Install necessary tools

Before we begin, you have to install the required tools to use git-buildpackage:

```
$ sudo apt-get install git-buildpackage dh-make quilt
```

Import your existing package or create a new package from scratch

Import existing package

In order to utilize the package made in HowToDebianWebApplicationPackaging you have to import the to git. This can be achieved by issuing the following command:

```
$ git-import-dsc /path/to/your/package/testpackage_0.1-1.dsc --pristine-tar
```

Use the *--pristine-tar* flag to create a pristine-tar branch to your repository. This enables you to generate a pristine upstream tarball later.

Create a new package from an upstream tarball

You can create a new package from an upstream tarball by using git-import-orig

```
$ cd testpackage
$ git branch
```

You'll see a list of the branches in your repository, by default you will have only 2 branches: **master** and **upstream**, but since we used the *--pristine-tar* flag we will have a 3rd one as well. It is called **pristine-tar** git-buildpackage requires that the upstream code and Debian/Ubuntu modified code are separated. **upstream** branch
contains the upstream code and **master** branch contains the Debian/Ubuntu modifications.The name of the branches can be changed in git-buildpackage's configuration. But it will not be covered here.

All modifications should be done to a special patch branch, but we will return to that later.
Your package is now ready, but we will do a few more tweaks to demonstrate git-buildpackage's features.

Configuration files

In order to save a bit of typing it is a good idea to save the a few command line arguments to git-buildpackage's configuration as defaults.
The configuration file is called .gbp.conf and can be located either in your home directory or in the git repository's

root directory. Saving the file to the git repository's root directory is a good idea because the configuration can be shared by all of the developers. Insert the following into either ~/.gbp.conf if /PATH/TO/YOUR/PACKAGE/.gbp.conf

```
[git-buildpackage]
export-dir=../build-area
pristine-tar=True
```

the export-dir directive defines path to a directory where the package will be built. It is a good idea to separate it from the package directory so you won't accidentally commit any build files to version control. The pristine-tar directive means that pristine-tar will be used for all tar operations by default.

It is also a good idea to add .pc folder to .gitignore. This makes sure that generated patch files won't pollute the version control system

```
$ echo ".pc" >> .gitignore
```

Finally a few modifications to configuration files found in the debian/ directory.

```
$ echo "unapply-patches" >> debian/source/local-options
$ echo "3.0 (quilt)" >> debian/source/format
```

The unapply-patches directive unapplies the patches after building the package, keeping your development branch clean.
"3.0 (quilt)" let's the debian package building scripts know that this package uses quilt to manage it's patches. **If you're going to use quilt to manage your patches your package will also have to depend on quilt.** This most likely something that you want to do.

Modifications as patches

Now that the package has been imported, it is a good idea to separate the modifications made by us from the upstream
code. A tool called gbp-pq(Git Build Package-Patch Queue) is able to help with this. gbp-pq is used to generate a patch-queue branch, where all commits will be generated into a patch. gbp-pq will create this branch for you if you issue the following command:

```
$ gbp-pq import
```

The generated branch will be called patch-queue/YOURCURRENTBRANCH so if you're in the master branch when executing the export command you will end up with a branch called patch-queue/master. gbp-pq will also checkout to the created branch. Now you can do your modifications in the patch-queue branch. A single commit will be generated into a single patch file, you should keep this in mind while doing modifications. If you happen to make silly mistakes you can always rearrange your branch's history using the normal git commands for that(rebase, squash etc).

After you have made your modifications and are happy with your commits checkout to master branch and execute gbp-pq export:

```
$ git checkout master
$ gbp-pq export
```

This will generate a series of quilt patches from your patch-queue/master branch and make sure that they get applied when you build
the package.

You need to also beware that any modifications you make to branch master after the patch-queue/master branch has been generated will have to be transferred to the patch-queue branch by executing:

```
$ gbp-pq rebase
```

If you don't rebase your patch-queue branch after modifications to the debian(in this case master) branch the modifications will be generated into patches. You most probably do not want this to happen.

Create a patch

Make sure that you are in the patch-queue/master branch and modify the index.php file of testpackage to contain the following:

```
<?php
    print("Package specific modification");
    phpinfo();
?>
```

Add and commit your changes

```
$ git add index.php
$ git commit -m "Added a package specific modification"
```

You can now import your patch to the master branch:

```
$ git checkout master
$ gbp-pq import
```

This will will generate a debian/patches/ folder with the commits made to patch-queue/master as patches.

You can commit the newly generated patch files to git if you want to and build your package using

```
$ git-buildpackage
```

If you don't want to commit the patch files to git you can build your package using

```
$ git-buildpackage --git-ignore-new
```

If you have set the build-area directive in your .gbp.conf the package can be found the folder specified by build-area(../build-area in the example configuration)

You can then install the package as usual with

```
$ sudo dpkg -i /PATH/TO/YOUR/PACKAGE
```

Miscellaneous

Name of the debian branch can be changed with the --git-debian-branch command line argument.
This is the branch from which the package will be built. It is also possible to specify this branch
the configuration files. like this:

```
[DEFAULT]
debian-branch = somebranch
upstream-branch = someotherbranch
```

If you add the directives to the DEFAULT section of the configuration file, the settings will be applied
for all tools of git-buildpackage(git-import-orig, git-import-dsc, etc). You can also specify the settings
specifically for individual tools by defining them in their respective sections. For example to set git-import-orig's
default debian branch to "1.3-dev" insert the following to your .gbp.conf file:

```
[git-import-orig]
debian-branch = 1.3-dev
```

References

http://raphaelhertzog.com/2010/11/18/4-tips-to-maintain-a-3-0-quilt-debian-source-package-in-a-vcs/
https://honk.sigxcpu.org/piki/development/debian_packages_in_git/
http://honk.sigxcpu.org/projects/git-buildpackage/manual-html/gbp.html

Appendix 4 Using reprepro

About this How To

This how to describes how to set up a package repository for Debian packages using reprepro

Install reprepro

```
$ sudo apt-get install reprepro
```

Create reprepro directory

Create a directory that will contain your repository. This directory will contain all configuration files etc. In this example we'll use /var/packages as the reprepro directory.

```
$ sudo mkdir /var/packages
```

This directory could also contains repositories for different distributions. It is a good idea to create a distribution specific subdirectory

```
$ cd /var/packages
$ sudo  mkdir ubuntu
```

After the main directory has been created a few more directories are needed. These directories are:
conf dists incoming indices logs pool project tmp

```
$ cd /var/packages/ubuntu
$ sudo mkdir conf dists incoming indices logs pool project tmp
```

Configuration

We will need 3 files to define the configuration for reprepro.

- distributions - this file contains information about supported distributions and their versions. You can define multiple repositories for different distributions by separating the configuration clauses by a whitespace.
- incoming - This file contains configuration concerning the incoming package queue.
- uploaders - This file defines who is allowed to upload packages to the repository. This can be done using Unix file system restrictions or by gpg.

distributions file

Let's start with the distributions file. Let's define a repository for Ubuntu 10.04 that will contain FreeNEST packages. Add this to the file

```
Origin: FreeNEST
Label: FreeNEST
Suite: unstable
Codename: lucid
Version: 10.04
```

```
Architectures: i386 amd64 source
Components: main
Description: FreeNEST package repository.
DebOverride: ../indices/override.lucid.main
UDebOverride: ../indices/override.lucid.main.debian-installer
DscOverride: ../indices/override.hardy.main.src
DebIndices: Packages Release . .gz .bz2
DscIndices: Sources Release .gz .bz2
Contents: . .gz .bz2
Log: freenestrepo.log
```

Origin Optional field that is copied into the release files.

Label same as Origin

Codename is a required field and defines an unique identifier of a distribution. It also used as directory name within dists/.

Suite This is an optional field that can be set to stable, unstable or testing for example.

Version Optional field that is once again simply copied into the Release files.

Architectures Required field that defines supported binary architectures within this distribution.

Component Required field. Defines the components of a distribution (eg. main)

see reprepro man page for more detailed information about the distributions file

```
$ man reprepro
```

incoming file

Next we'll write the incoming file. Insert the following to the file:

```
Name: default
IncomingDir: incoming
TempDir: tmp
Allow: lucid lucid-backports
Cleanup: on_deny on_erro
```

Name defines a name for this ruleset.

IncomingDir defines the folder from which to scan for .changes files.

TempDir path to the directory what will contain temporary files during processing of packages.

Cleanup if something goes wrong while processing the package delete it and all files it references.

As usual the reprepro man page contains more detailed infromation about the incoming file and it's configuration.

uploaders file

Insert this to the uploaders file

```
allow * by unsigned
```

This allows unsigned packages to be uploaded to the repository. More detailed information can be found from the reprepro man page.

Web server configuration

You have to make your repository folder accessable so people can use it. You can do this by creating a symlink from your webserver's document root to /var/packages. Let's call our symlink packages.

```
$ cd /var/www
$ sudo ln -s packages /var/packages
```

Now you can access your repository through your browser: http://url-to-your-repository/packages

It is a good idea to hide some reprepro folders(db, conf, incoming) from public view. You can achieve this on Apache by
adding the following lines to /etc/apache2/conf.d/packages

```
<Directory "/var/www/packages/">
        Options Indexes FollowSymLinks Multiviews
        Order allow,deny
        Allow from all
</Directory>

<Directory "/var/www/packages/*/db/">
        Order allow,deny
        Deny from all
</Directory>

<Directory "/var/www/packages/*/conf/">
        Order allow,deny
        Deny from all
</Directory>

<Directory "/var/www/packages/*/incoming/">
        Order allow,deny
        Deny from all
</Directory>
```

The wildcard chracter(*) in the directory definition makes sure that directory gets blocked in
all subdirectories. Eg. /var/www/packages/ubuntu/incoming and /var/www/packages/debian/incoming
get blocked with the same directive.

Adding packages to the repository

You can add packages manually to the repository using the following command:

```
$ reprepro includedb osversion pathtopackage
```

So if we wanted to add a package called ./freenestbase_0.1-i386.deb to lucid repository we would use:

```
$ cd /var/packages/ubuntu
$ reprepro includedeb lucid ./freenestbase_0.1-i386.deb
```

Removing packages from the repository

You can remove packages with reprepro's remove command

```
$ reprepro remove osversion packagename
```

So if we want to remove the package we added earlier, we could use:

```
$ sudo reprepro remove lucid freenestbase
```

Using the repository

You have to add the repository to your sources.list file in order to use the repository
Insert the following to your /etc/apt/sources.list.

```
deb http://url-to-your-repository/packages/ubuntu lucid main
```

and run

```
$ sudo apt-get update
```

Now you can install the previously added freenestbase package by typing:

```
$ sudo apt-get install freenestbase
```